Javascript

Anything about Javascript in general, that isn't specific to Node.js.

- Whirlwind tour of (correct) npm usage
- An overview of Javascript tooling
- Monolithic vs. modular - what's the difference?
- Synchronous vs. asynchronous
- What is state?
- Promises reading list
- The Promises FAQ - addressing the most common questions and misconceptions about Promises
- Error handling (with Promises)
- Bluebird Promise.try using ES6 Promises
- Please don't include minified builds in your npm packages!
- How to get the actual width of an element in jQuery, even with border-box: box-sizing
- A survey of unhandledRejection and rejectionHandled handlers
- Quill.js glossary
- Riot.js cheatsheet
- Quick reference for `checkit` validators
- ES Modules are terrible, actually
- A few notes on the "Gathering weak npm credentials" article

Whirlwind tour of (correct) npm usage

> This article was originally published at https://gist.github.com/joepie91/9b9dbd8c9ac3b55a65b2.

This is a quick tour of how to get started with NPM, how to use it, and how to fix it.

> I'm available for tutoring and code review :)

Starting a new project

Create a folder for your project, preferably a Git repository. Navigate into that folder, and run:

```
npm init
```

It will ask you a few questions. Hit `Enter` without input if you're not sure about a question, and it will use the default.

You now have a `package.json`.

> *If you're using Express:* Please don't use `express-generator`. It sucks. Just use `npm init` like explained above, and follow the 'Getting Started' and 'Guide' sections on the Express website. They will teach you all you need to know when starting from scratch.

Installing a package

All packages in NPM are *local* - that is, specific to the project you install it in, and actually installed *within* that project. They are also nested - if you use the `foo` module, and `foo` uses the `bar` module, then you will have a `./node_modules/foo/node_modules/bar`. This means you pretty much never have version conflicts, and can install as many modules as you want without running into issues.

All modern versions of NPM will 'deduplicate' and 'flatten' your module folder as much as possible to save disk space, but as a developer you don't have to care about this - it will still work like it's a

tree of nested modules, and you can still assume that there will be no version conflicts.

You install a package like this:

```
npm install packagename
```

While the packages themselves are installed in the `node_modules` directory (as that's where the Node.js runtime will look for them), that's only a temporary install location. The *primary* place where your dependencies are defined, should be in your `package.json` file - so that they can be safely updated and reinstalled later, even if your `node_modules` gets lost or corrupted somehow.

In older versions of npm, you had to manually specify the `--save` flag to make sure that the package is saved in your `package.json`; that's why you may come across this in older articles. However, modern versions of NPM do this automatically, so the command above should be enough.

One case where you *do* still need to use a flag, is when you're installing a module that you just need for developing your project, but that isn't needed when actually *using* or *deploying* your project. Then you can use the `--save-dev` flag, like so:

```
npm install --save-dev packagename
```

Works pretty much the same, but saves it as a development dependency. This allows a user to install just the 'real' dependencies, to save space and bandwidth, if they just want to use your thing and not modify it.

To install everything that is declared in `package.json`, you just run it without arguments:

```
npm install
```

When you're using Git or another version control system, you should add `node_modules` to your ignore file (eg. `.gitignore` for Git); this is because *installed* copies of modules may need to be different depending on the system. You can then use the above command to make sure that all the dependencies are correctly installed, after cloning your repository to a new system.

Semantic versioning

Packages in NPM usually use semantic versioning; that is, the changes in a version number indicate what has changed, and whether the change is breaking. Let's take **1.2.3** as an example version. The components of that version number would be:

- **Major version number:** 1
- **Minor version number:** 2
- **Patch version number:** 3

Depending on which number changes, there's a different kind of change to the module:

- **Patch version upgrade (eg. `1.2.3` -> `1.2.4`):** An internal change was made, but the API hasn't changed. It's safe to upgrade.
- **Minor version upgrade (eg. `1.2.3` -> `1.3.0`):** The API has changed, but in a backwards-compatible manner - for example, a new feature or option was added. It's safe to upgrade. You may still want to read the changelog, in case there's new features that you want to use, or that you were waiting for.
- **Major version upgrade (eg. `1.2.3` -> `2.0.0`):** The API has changed, and is not backwards-compatible. For example, a feature was removed, a default was changed, and so on. It is **not** safe to upgrade. You first need to read the changelog, to see whether the changes affect your application.

Most NPM packages follow this, and it gives you a lot of certainty in what upgrades are safe to carry out, and what upgrades aren't. NPM explicitly adopts semver in its package.json as well, by introducing a few special version formats:

- **`~1.2.3`:** Allow automatic patch upgrades, but not minor or major upgrades. Upgrading to `1.2.4` is allowed, but upgrading to `1.3.0` or `2.0.0` is not. You still can't downgrade below `1.2.3` - for example, `1.2.2` is *not* allowed.
- **`^1.2.3`:** Allow automatic patch and minor upgrades, but not major upgrades. Upgrading to `1.2.4` or `1.3.0` is allowed, but upgrading to `2.0.0` is not. You still can't downgrade below `1.2.3` - for example, `1.2.2` or `1.1.0` are *not* allowed.
- **`1.2.3`:** Require this specific version. No upgrades are allowed. You will rarely need this - only for misbehaving packages, really.
- **`*`:** Allow upgrades to whatever the latest version is. You should **never** use this.

By default, NPM will automatically use the **^1.2.3** notation, which is usually what you want. Only configure it otherwise if you have an explicit reason to do so.

A special case are `0.x.x` versions - these are considered to be 'unstable', and the rules are slightly different: the *minor* version number indicates a breaking change, rather than the major version number. That means that `^0.1.2` will allow an upgrade to `0.1.3`, but *not* to `0.2.0`. This is commonly used for pre-release testing versions, where things may wildly change with every release.

If you end up publishing a module yourself (and you most likely eventually will), then definitely adhere to these guidelines as well. They make it a lot easier for developers to keep dependencies up to date, leading to considerably less bugs and security issues.

Global modules

Sometimes, you want to install a command-line utility such as `peerflix`, but it doesn't belong to any particular project. For this, there's the `--global` or `-g` flag:

```
npm install -g peerflix
```

If you used packages from your distribution to install Node, you may have to use `sudo` for global modules.

Never, ever, ever use global modules for project dependencies, ever. It may seem 'nice' and 'efficient', but you will land in dependency hell. It is not possible to enforce semver constraints on global modules, and things will spontaneously break. All the time. Don't do it. Global modules are *only for project-independent, system-wide, command-line tools*.

This applies even to development tools for your project. Different projects will often need different, incompatible versions of development tools - so those tools should be installed *without* the global flag. For local packages, the binaries are all collected in `node_modules/.bin`. You can then run the tools like so:

```
./node_modules/.bin/eslint
```

NPM is broken, and I don't understand the error!

The errors that NPM shows are usually not very clear. I've written a tool that will analyze your error, and try to explain it in plain English. It can be found here.

My dependencies are broken!

If you've just updated your Node version, then you may have native (compiled) modules that were built against the old Node version, and that won't work with the new one. Run this to rebuild them:

```
npm rebuild
```

My dependencies are still broken!

Make sure that all your dependencies are declared in `package.json`. Then just remove and recreate your `node_modules`:

```
rm -rf node_modules
npm install
```

An overview of Javascript tooling

> This article was originally published at https://gist.github.com/joepie91/3381ce7f92dec7a1e622538980c0c43d.

Getting confused about the piles of development tools that people use for Javascript? Here's a quick index of what is used for what.

Keep in mind that you shouldn't add tools to your workflow for the sake of it. While you'll see many production systems using a wide range of tools, these tools are typically used because they solved a *concrete problem* for the developers working on it. You should **not** add tools to your project unless you have a concrete problem that they can solve; none of the tools here are *required*.

Start with nothing, and add tools as needed. This will keep you from getting lost in an incomprehensible pile of tooling.

Build/task runners

Typical examples: Gulp, Grunt

These are not exactly build tools in and of themselves; they're rather just used to glue together *other* tools. For example, if you have a set of build steps where you need to run tool A after tool B, a build runner can help to orchestrate those tools.

Bundlers

Typical examples: Browserify, Webpack, Parcel

These tools take a bunch of `.js` files that use modules (either CommonJS using `require()` statements, or ES Modules using `import` statements), and combine them into a *single* `.js` file. Some of them also allow specifying 'transformation steps', but their main purpose is bundling.

Why does bundling matter? While in Node.js you have access to a module system that lets you load files as-needed from disk, this wouldn't be practical in a browser; fetching every file individually over the network would be very slow. That's why people use a bundler, which effectively does all this work upfront, and then produces a single 'combined' file with all the same guarantees of a module system, but that can be used in a browser.

Bundlers can also be useful for running module-using code in very basic JS environments that don't have module support for some reason; this includes Google Sheets, extensions for PostgreSQL, GNOME, and so on.

Bundlers are *not* transpilers. They do not compile one language to another, and they don't "make ES6 work everywhere". Those are the job of a *transpiler*. Bundlers are sometimes configured to *use* a transpiler, but the transpiling itself isn't done by the bundler.

Bundlers are *not* task runners. This is an especially popular misconception around Webpack. Webpack does *not* replace task runners like Gulp; while Gulp is designed to glue together arbitrary build tasks, Webpack is specifically designed for *browser bundles*. It's commonly useful to use Webpack *with* Gulp or another task runner.

Transpilers

Typical examples: Babel, the TypeScript compiler, CoffeeScript

These tools take a bunch of code in one language, and 'compile' it to another language. They're called commonly 'transpilers' rather than 'compilers' because unlike traditional compilers, these tools don't compile to a lower-level representation; they're just different languages at a similar level of abstraction.

These are typically used to run code written against newer JS versions in older JS runtimes (eg. Babel), or to provide custom languages with more conveniences or constraints that can then be executed in any regular JS environment (TypeScript, CoffeeScript).

Process restarters

Typical examples: nodemon

These tools automatically restart your (Node.js) process when the underlying code is changed. This is used for development purposes, to remove the need to manually restart your process every change.

A process restarter may either watch for file changes itself, or be controlled by an external tool like a build runner.

Page reloaders

Typical examples: LiveReload, BrowserSync, Webpack hot-reload

These tools automatically refresh a page in the browser and/or reload stylesheets and/or re-render parts of the page, to reflect the changes in your *browser-side* code. They're kind of the equivalent of a process restarter, but for webpages.

These tools are usually externally controlled; typically by either a build runner or a bundler, or both.

Debuggers

Typical examples: Chrome Developer Tools, node-inspect

These tools allow you to inspect *running* code; in Node.js, in your browser, or both. Typically they'll support things like pausing execution, stepping through function calls manually, inspecting variables, profiling memory allocations and CPU usage, viewing execution logs, and so on.

They're typically used to find tricky bugs. It's a good idea to learn how these tools work, but often it'll still be easier to find a bug by just 'dumb logging' variables throughout your code using eg. `console.log`.

Monolithic vs. modular - what's the difference?

> This article was originally published at https://gist.github.com/joepie91/7f03a733a3a72d2396d6.

When you're developing in Node.js, you're likely to run into these terms - "monolithic" and "modular". They're usually used to describe the different types of frameworks and libraries; not just HTTP frameworks, but modules in general.

At a glance

- **Monolithic:** "Batteries-included" and typically tightly coupled, it tries to include all the stuff that's needed for common usecases. An example of a monolithic web framework would be Sails.js.
- **Modular:** "Minimal" and loosely coupled. Only includes the bare minimum of functionality and structure, and the rest is a plugin. Fundamentally, it generally only has a single 'responsibility'. An example of a modular web framework would be Express.

Coupled?

In software development, the terms "tightly coupled" and "loosely coupled" are used to indicate how much components rely on each other; or more specifically, how many assumptions they make about each other. This directly translates to how easy it is to replace and change them.

- **Tightly coupled:** Highly cohesive code, where every part of the code makes assumptions about every other part of the code.
- **Loosely coupled:** Very "separated" code, where every part of the code communicates with other parts through more-or-less standardized and neutral interfaces.

While tight coupling can sometimes result in slightly more performant code and very occasionally makes it easier to build a 'mental model', loosely coupled code is much easier to understand and maintain - as the inner workings of a component are separated from its interface or API, you can make many more assumptions about how it behaves.

Loosely coupled code is often centered around 'events' and data - a component 'emits' changes that occur, with data attached to them, and other components may optionally 'listen' to those events and do something with it. However, the emitting component has no idea who (if anybody!)

is listening, and cannot make assumptions about what the data is going to be used for.

What this means in practice, is that loosely coupled (and modular!) code rarely needs to be changed - once it is written, has a well-defined set of events and methods, and is free of bugs, it no longer needs to change. If an application wants to start using the data differently, it doesn't require changes in the component; the data is still of the same format, and the application can simply process it differently.

This is only one example, of course - loose coupling is more of a practice than a pattern. The exact implementation depends on your usecase. A quick checklist to determine how loosely coupled your code is:

- ☐ **Does your component rely on external state?** This is an absolute no-no. Your component cannot rely on any state outside of the component itself. It may not make any assumptions about the application *whatsoever*. Don't even rely on configuration files or other filesystem files - all such data must be passed in by the application explicitly, always. What isn't in the component itself, doesn't exist.
- ☐ **How many assumptions does it make about how the result will be used?** Loosely coupled code shouldn't care about how its output will be used, whether it's a return value or an event. The output just needs to be consistent, documented, and neutral.
- ☐ **How many custom 'types' are used?** Loosely coupled code should generally only accept objects that are defined on a language or runtime level, and in common use. Arrays and A+ promises are fine, for example - a proprietary representation of an ongoing task is not.
- ☐ **If you *need* a custom type, how simple is it?** If absolutely needed, your custom object type should be as plain as possible - just a plain Javascript object, optimally. It should be well-documented, and not duplicate an existing implementation to represent this kind of data. Ideally, it should be defined in a separate project, just for documenting the type; that way, others can implement it as well.

In this section, I've used the terms "component" and "application", but these are interchangeable with "callee"/"caller", and "provider"/"consumer". The principles remain the same.

The trade-offs

At first, a monolithic framework might look easier - after all, it already includes everything you think you're going to need. In the long run, however, you're likely to run into situations where the framework just doesn't *quite* work how you want it to, and you have to spend time trying to work around it. This problem gets worse if your usecase is more unusual - because the framework developers didn't keep in mind your usecase - but it's a risk that always exists to some degree.

Initially, a modular framework might look harder - you have to figure out what components to use for yourself. That's a one-time cost, however; the majority of modules are reusable across projects, so after your first project you'll have a good idea of what to start with. The remaining usecase-

specific modules would've been just as much of a problem in a monolithic framework, where they likely wouldn't have existed to begin with.

Another consideration is the possibility to 'swap out' components. What if there's a bug in the framework that you're unable (or not allowed) to fix? When building your application modularly, you can simply get rid of the offending component and replace it with a different one; this usually doesn't take more than a few minutes, because components are typically small and only do one thing.

In a monolithic framework, this is more problematic - the component is an inherent part of the framework, and replacing it may be impossible or extremely hard, depending on how many assumptions the framework makes. You will almost certainly end up implementing a workaround of some sort, which can take *hours*; you need to understand the framework's codebase, the component you're using, and the exact reason why it's failing. Then you need to write code that works around it, sometimes even having to 'monkey-patch' framework methods.

Relatedly, you may find out halfway through the project that the framework doesn't support your usecase as well as you thought it would. Now you have to either *replace the entire framework*, or build hacks upon hacks to make it 'work' somehow; well enough to convince your boss or client, anyway. The higher cost for on-boarding new developers (as they have to learn an entire framework, not just the bits you're interested in *right now*), only compounds this problem - now they *also* have to learn why all those workarounds exist.

In summary, the tradeoffs look like this:

- **Monolithic:** Slightly faster to get started with, but less control over its workings, more chance of the framework not supporting your usecase, and higher long-term maintenance cost due to the inevitable need for workarounds.
- **Modular:** Takes slightly longer to get started on your first project, but total control over its workings, practically every usecase is supported, and long-term maintenance is cheaper.

The "it's just a prototype!" argument

When explaining this to people, a common justification for picking a monolithic framework is that "it's just a prototype!", or "it's just an MVP!", with the implication that it can be changed later. In reality, it usually can't.

Try explaining to your boss that you want to throw out the working(!) code you have, and rewrite everything from the ground up in a different, more maintainable framework. The best response that you're likely to get, is your boss questioning why you didn't use that framework to begin with - but more likely, the answer is "no", and you're going to be stuck with your hard-to-maintain monolithic codebase for the rest of the project or your employment, whichever terminates first.

Again, the cost of a modular codebase is a one-time cost. After your first project, you already know where to find most modules you need, and building on a modular framework will not be more

expensive than building on a monolithic one. Don't fall into the "prototype trap", and do it right from day one. You're likely to be stuck with it for the rest of your employment.

Synchronous vs. asynchronous

> This article was originally published at https://gist.github.com/joepie91/bf3d04febb024da89e3a3e61b164247d.

You'll run into the terms "synchronous" and "asynchronous" a lot when working with JS. Let's look at what they actually *mean*.

Synchronous code is like what you might be used to already from other languages. You call a function, it does some work, and then returns the result. No other code runs in the meantime. This is simple to understand, but it's also inefficient; what if "doing some work" mostly involves getting some data from a database? In the meantime, our process is sitting around doing nothing, waiting for the database to respond. It could be doing useful work in that time!

And that's what brings us to asynchronous code. Asynchronous code works differently; you still call a function, but it *doesn't* return a result. Instead, you don't just pass the regular arguments to the function, but also give it a piece of code in a function (a so-called "asynchronous callback") to execute when the operation completes. The JS runtime stores this callback alongside the in-progress operation, to retrieve and execute it later when the external service (eg. the database) reports that the operation has been completed.

Crucially, this means that when you call an asynchronous function, it *cannot* wait until the external processing is complete before returning from the function! After all, the intention is to keep running other code in the meantime, so it needs to return from the function so that the 'caller' (the code which originally called the function) can continue doing useful things even while the external operation is in progress.

All of this takes place in what's called the "event loop" - you can pretty much think of it as a huge infinite loop that contains your entire program. Every time you trigger an external process through an asynchronous function call, that external process will eventually finish, and put its result in a 'queue' alongside the callback you specified. On each iteration ("tick") of the event loop, it then goes through that queue, executes all of the callbacks, which can then indirectly cause new items to be put into the queue, and so on. The end result is a program that calls asynchronous callbacks as and when necessary, and that keeps giving new work to the event loop through a chain of those callbacks.

This is, of course, a very simplified explanation - just enough to understand the rest of this page. I strongly recommend reading up on the event loop more, as it will make it much easier to understand JS in general. Here are some good resources that go into more depth:

1. https://nodesource.com/blog/understanding-the-nodejs-event-loop (article)

2. https://www.youtube.com/watch?v=8aGhZQkoFbQ (video)
3. https://www.youtube.com/watch?v=cCOL7MC4Pl0 (video)

Now that we understand the what the event loop is, and what a "tick" is, we can define more precisely what "asynchronous" means in JS:

Asynchronous code is code that happens across more than one event loop tick. An asynchronous function is a function that needs more than one event loop tick to complete.

This definition will be important later on, for understanding why asynchronous code can be more difficult to write correctly than synchronous code.

Asynchronous execution order and boundaries

This idea of "queueing code to run at some later tick" has consequences for how you write your code.

Remember how the event loop is a loop, and ticks are iterations - this means that event loop ticks are *distributed across time linearly*. First the first tick happens, then the second tick, then the third tick, and so on. Something that runs in the first tick can *never* execute before something that runs in the third tick; unless you're a time traveller anyway, in which case you probably would have more important things to do than reading this guide □□

Anyhow, this means that code will run in a slightly counterintuitive way, if you're used to synchronous code. For example, consider the following code, which uses the asynchronous `setTimeout` function to run something after a specified amount of milliseconds:

```
console.log("one");

setTimeout(() => {
□console.log("two");
}, 300);

console.log("three");
```

You might expect this to print out `one, two, three` - but if you try running this code, you'll see that it doesn't! Instead, you get this:

```
one
three
two
```

What's going on here?!

The answer to that is what I mentioned earlier; the asynchronous callback is *getting queued for later*. Let's pretend for the sake of explanation that an event loop tick only happens when there's actually something to do. The **first tick** would then run this code:

```
console.log("one");

setTimeout(..., 300); // This schedules some code to run in a next tick, about 300ms later

console.log("three");
```

Then 300 milliseconds elapse, with nothing for the event loop to do - and after those 300ms, the callback we gave to `setTimeout` suddenly appears in the event loop queue. Now the **second tick** happens, and it executes this code:

```
console.log("two");
```

... thus resulting in the output that we saw above.

The key insight here is that **code with callbacks does not execute in the order that the code is written**. Only the code *outside* of the callbacks executes in the written order. For example, we can be certain that `three` will get printed after `one` because both are outside of the callback and so they are executed in that order, but because `two` is printed from *inside* of a callback, we can't know when it will execute.

"*But hold on*", you say, "*then how can you know that `two` will be printed after `three` and `one`?*"

This is where the earlier definition of "asynchronous code" comes into play! Let's reason through it:

1. `setTimeout` is asynchronous.
2. Therefore, we call `console.log("two")` from within an asynchronous callback.
3. Synchronous code executes within one tick.
4. Asynchronous code needs more than one tick to execute, ie. the asynchronous callback will be called in a *later* tick than the one where we started the operation (eg. `setTimeout`).
5. Therefore, an asynchronous callback will *always* execute after the synchronous code that started the operation, no matter what.
6. Therefore, `two` will *always* be printed after `one` and `three`.

So, we can know when the asynchronous callback will be executed, in terms of relative time. That's useful, isn't it? Doesn't that mean that we can do that for *all* asynchronous code? Well, unfortunately not - it gets more complicated when there is *more* than one asynchronous operation.

Take, for example, the following code:

```
console.log("one");

someAsynchronousOperation(() => {
  console.log("two");
});

someOtherAsynchronousOperation(() => {
  console.log("three");
});

console.log("four");
```

We have two different asynchronous operations here, and we don't know for certain which of the two will finish faster. We don't even know whether it's always the *same* one that finishes faster, or whether it varies between runs of the program. So while we can determine that `two` and `three` will *always* be printed after `one` and `four` - remember, asynchronous callbacks in synchronous code - we *can't* know whether `two` or `three` will come first.

And this is, fundamentally, what makes asynchronous code more difficult to write; you never know for sure in what order your code will complete. Every real-world program will have at least *some* scenarios where you can't force an order of operations (or, at least, not without horribly bad performance), so this is a problem that you *have* to account for in your code.

The easiest solution to this, is to avoid "shared state". Shared state is information that you store (eg. in a variable) and that gets used by multiple parts of your code independently. This can sometimes be necessary, but it also comes at a cost - if function A and function B both modify the same variable, then if they run in a different order than you expected, one of them might mess up the expected state of the other. This is generally already true in programming, but even more important when working with asynchronous code, as your chunks of code get 'interspersed' much more due to the callback model.

[...]

What is state?

> This article was originally published at https://gist.github.com/joepie91/8c2cba6a3e6d19b275fdff62bef98311.

"State" is data that is associated with some part of a program, and that can be changed over time to change the behaviour of the program. It doesn't have to be changed by the user; it can be changed by *anything* in the program, and it can be *any* kind of data.

It's a bit of an abstract concept, so here's an example: say you have a button that increases a number by 1 every time you click it, and the (pseudo-)code looks something like this:

```
let counter = 0;
let increment = 1;

button.on("click", () => {
	counter = counter + increment;
});
```

In this code, there are two bits of "state" involved:

1. **Whether the button is clicked:** This bit of data - specifically, the change between "yes" and "no" - is what determines when to increase the counter. The example code doesn't interact with this data directly, but the callback is called whenever it changes from "no" to "yes" and back again.
2. **The current value of the counter:** This bit of data is used to determine what the *next* value of the counter is going to be (the current value plus one), as well as what value to show on the screen.

Now, you may note that we also define an `increment` variable, but that it isn't in the list of things that are "state"; this is because the `increment` value *never changes*. It's just a static value (`1`) that is always the same, even though it's stored in a variable. That means it's *not* state.

You'll also note that "whether the button is clicked" isn't stored in any variable we have access to, and that we can't access the "yes" or "no" value directly. This is an example of what we'll call *invisible state* - data that *is* state, but that we cannot see or access directly - it only exists "behind the scenes". Nevertheless, it still affects the behaviour of the code through the event handler callback that we've defined, and that means it's still state.

Promises reading list

> This article was originally published at https://gist.github.com/joepie91/791640557e3e5fd80861.

This is a list of examples and articles, in roughly the order you should follow them, to show and explain how promises work and why you should use them. I'll probably add more things to this list over time.

This list primarily focuses on Bluebird, but the basic functionality should also work in ES6 Promises, and some examples are included on how to replicate Bluebird functionality with ES6 promises. You should still use Bluebird where possible, though - they are faster, less error-prone, and have more utilities.

> I'm available for tutoring and code review :)

If you get stuck

I've made a brief FAQ of common questions that people have about Promises, and how to use them. If you don't understand something listed here, or you're wondering how to implement a specific requirement, chances are that it'll be answered in that FAQ.

Compatibility

Bluebird will **not** work correctly (in client-side code) in older browsers. If you need to support older browsers, and you're using Webpack or Browserify, you should use the `es6-promise` module instead, and reimplement behaviour where necessary.

Introduction

- Start reading here, to understand why Promises matter.
- If it's not quite clear yet, some code that uses callbacks, and its equivalent using Bluebird.
- A demonstration of how promise chains can be 'flattened'

Promise.try

Many guides and examples fail to demonstrate Promise.try, or to explain why it's important. This article will explain it.

Error handling

- A quick introduction
- An illustration of error bubbling: step 1, step 2
- Implementing 'fallback' values (ie. defaults for when an asynchronous operation fails)
- bluebird-tap-error, a module for intercepting and looking at errors, without preventing propagation. Useful if you need to do the actual error handling elsewhere.
- Handling errors in Express, using Promises

Many examples on the internet don't show this, but you should **always** start a chain of promises with Promise.try, and if it is within a function or callback, you should always **return** your promises chain. Not doing so, will result in less reliable error handling and various other issues (eg. code executing too soon).

Promisifying

- Promisifying functions and modules that use nodebacks (Node.js callbacks)
- An example of manually promisifying an EventEmitter
- Promisifying `fs.exists` (which is async, but doesn't follow the nodeback convention)

Functional (map, filter, reduce)

- Functional programming in Javascript: map, filter and reduce (an introduction, not Bluebird-specific, but important to understand)
- (Synchronous) examples of map, filter, and reduce in Bluebird
- Example of using map for retrieving a (remote) list of URLs with bhttp

Nesting

- Example of retaining scope through nesting
- Example of 'breaking out' of a chain through nesting
- Example of a nested Promise.map

- An example with increasing complexity, implementing an 'error-tolerant' Promise.map: part 1, part 2, part 3

ES6 Promises

- Documentation on MDN
- Promise.try using ES6 Promises
- Promise.delay using ES6 Promises

Odds and ends

Some potentially useful snippets:

- Flattening an array of arrays, when using promises

You're unlikely to need any of these things, if you just stick with either Bluebird or ES6 promises:

- How to test whether a Promises implementation handles callbacks correctly
- Why this matters.

The Promises FAQ - addressing the most common questions and misconceptions about Promises

> This article was originally published at https://gist.github.com/joepie91/4c3a10629a4263a522e3bc4839a28c83. Nowadays Promises are more widely understood and supported, and it's not as relevant as it once was, but it's kept here for posterity.

> By the way, I'm available for tutoring and code review :)

You'll find a table of contents on your left.

1. What Promises library should I use?

That depends a bit on your usecase.

My usual recommendation is Bluebird - it's robust, has good error handling and debugging facilities, is fast, and has a well-designed API. The downside is that Bluebird will not correctly work in older browsers (think Internet Explorer 8 and older), and when used in Browserified/Webpacked code, it can sometimes add a lot to your bundle size.

ES6 Promises are gaining a lot of traction purely because of being "ES6", but in practice they are just *not very good*. They are generally lacking standardized debugging facilities, they are missing essential utilities such as Promise.try/promisify/promisifyAll, they cannot catch specific error types (this is a big robustness issue), and so on.

ES6 Promises can be useful in constrained scenarios (eg. older browsers with a polyfill, restricted non-V8 runtimes, etc.) but I would not generally recommend them.

There are many other Promise implementations (Q, WhenJS, etc.) - but frankly, I've not seen any that are an improvement over either Bluebird or ES6 Promises in their respective 'optimal scenarios'. I'd also recommend explicitly *against* Q because it is extremely slow and has a very poorly designed API.

In summary: Use Bluebird, unless you have a very specific reason not to. In those very specific cases, you probably want ES6 Promises.

2. How do I create a Promise myself?

Usually, you don't. Promises are not usually something you 'create' explicitly - rather, they're a *natural consequence* of chaining together multiple operations. Take this example:

```
function getLinesFromSomething() {
    return Promise.try(() => {
        return bhttp.get("http://example.com/something.txt");
    }).then((response) => {
        return response.body.toString().split("\n");
    });
}
```

In this example, all of the following *technically* result in a new Promise:

- `Promise.try(...)`
- `bhttp.get(...)`
- The synchronous value from the `.then` callback, which gets converted automatically to a resolved Promise (see question 5)

... but none of them are explicitly created as "a new Promise" - that's just the natural consequence of starting a chain with `Promise.try` and then returning Promises or values from the callbacks.

There is one example to this, where you *do* need to explicitly create a new Promise - when converting a different kind of asynchronous API to a Promises API, and even then you only need to do this if `promisify` and friends don't work. This is explained in question 7.

3. How do I use `new Promise`?

You don't, usually. In almost every case, you either need Promise.try, or some kind of promisification method. Question 7 explains how you should do promisification, and when you *do* need `new Promise`.

But when in doubt, don't use it. It's very error-prone.

4. How do I resolve a Promise?

You don't, usually. Promises are not something you need to 'resolve' manually - rather, you should just *return* some kind of Promise, and let the Promise library handle the rest.

There's one exception here: when you're manually promisifying a strange API using `new Promise`, you need to call `resolve()` or `reject()` for a successful and unsuccessful state, respectively. Make sure to read question 3, though - you should almost never actually *use* `new Promise`.

5. But what if I want to resolve a synchronous result or error?

You simply `return` it (if it's a result) or `throw` it (if it's an error), from your `.then` callback. When using Promises, synchronously returned values are automatically converted into a *resolved Promise*, whereas synchronously thrown errors are automatically converted into a *rejected Promise*. You don't need to use `Promise.resolve()` or `Promise.reject()`.

6. But what if it's at the start of a chain, and I'm not in a `.then` callback yet?

Using Promise.try will make this problem not exist.

7. How do I make this non-Promises library work with Promises?

That depends on what kind of API it is.

- **Node.js-style error-first callbacks:** Use Promise.promisify and/or Promise.promisifyAll to convert the library to a Promises API. For ES6 Promises, use the es6-promisify and es6-promisify-all libraries respectively. In Node.js, `util.promisify` can also be used.
- **EventEmitters:** It depends. Promises are explicitly meant to represent an operation that succeeds or fails *precisely once*, so *most* EventEmitters cannot be converted to a Promise, as they will have *multiple* results. Some exceptions exist; for example, the `response` event when making a HTTP request - in these cases, use something like bluebird-events.
- **setTimeout:** Use `Promise.delay` instead, which comes with Bluebird.
- **setInterval:** Avoid `setInterval` entirely (this is why), and use a recursive `Promise.delay` instead.
- **Asynchronous callbacks with a single result argument, and no `err`:** Use promisify-simple-callback.

- **A different Promises library:** No manual conversion is necessary, as long as it is compliant with the Promises/A+ specification (and nearly every implementation is). Make sure to use Promise.try in your code, though.
- **Synchronous functions:** No manual conversion is necessary. Synchronous returns and throws are automatically converted by your Promises library. Make sure to use Promise.try in your code, though.
- **Something else not listed here:** You'll probably have to promisify it manually, using `new Promise`. Make sure to keep the code within `new Promise` as minimal as possible - you should have a function that *only* promisifies the API you intend to use, without doing *anything* else. All further processing should happen outside of `new Promise`, once you already have a Promise object.

8. How do I propagate errors, like with `if(err) return cb(err)`?

You don't. Promises will propagate errors automatically, and you don't need to do anything special for it - this is one of the benefits that Promises provide over error-first callbacks.

When using Promises, the *only* case where you need to `.catch` an error, is if you intend to handle it - and you should always *only* catch the types of error you're interested in.

These two Gists (step 1, step 2) show how error propagation works, and how to `.catch` specific types of errors.

9. How do I break out of a Promise chain early?

You don't. You use conditionals instead. Of course, specifically for *failure scenarios*, you'd still throw an error.

10. How do I convert a Promise to a synchronous value?

You can't. Once you write asynchronous code, all of the 'surrounding' code *also* needs to be asynchronous. However, you can just have a Promise chain in the 'parent code', and return the Promise from your own method.

For example:

```
function getUserFromDatabase(userId) {
    return Promise.try(() => {
```

```
        return database.table("users").where({id: userId}).get();
    }).then((results) => {
        if (results.length === 0) {
            throw new MyCustomError("No users found with that ID");
        } else {
            return results[0];
        }
    });
}

/* Now, to *use* that getUserFromDatabase function, we need to have another Promise chain: */

Promise.try(() => {
    // Here, we return the result of calling our own function. That return value is a Promise.
    return getUserFromDatabase(42);
}).then((user) => {
    console.log("The username of user 42 is:", user.username);
});
```

(If you're not sure what Promise.try is or does, this article will explain it.)

11. How do I save a value from a Promise outside of the callback?

You don't. See question 10 above - you need to use Promises "all the way down".

12. How do I access previous results from the Promise chain?

In some cases, you might need to access an *earlier* result from a chain of Promises, one that you don't have access to anymore. A simple example of this scenario:

```
'use strict';

// ...

Promise.try(() => {
    return database.query("users", {id: req.body.userId});
```

```
}).then((user) => {
    return database.query("groups", {id: req.body.groupId});
}).then((group) => {
    res.json({
        user: user, // This is not possible, because `user` is not in scope anymore.
        group: group
    });
});
```

This is a fairly simple case - the `user` query and the `group` query are completely independent, and they can be run at the same time. Because of that, we can use `Promise.all` to run them in parallel, and return a combined Promise for *both* of their results:

```
'use strict';

// ...

Promise.try(() => {
    return Promise.all([
        database.query("users", {id: req.body.userId}),
        database.query("groups", {id: req.body.groupId})
    ]);
}).spread((user, group) => {
    res.json({
        user: user, // Now it's possible!
        group: group
    });
});
```

Note that instead of `.then`, we use `.spread` here. Promises only support a *single* result argument for a `.then`, which is why a Promise created by `Promise.all` would resolve to an array of `[user, group]` in this case. However, `.spread` is a Bluebird-specific variation of `.then`, that will automatically "unpack" that array into multiple callback arguments. Alternatively, you can use ES6 object destructuring to accomplish the same.

Now, the above example assumes that the two asynchronous operations are *independent* - that is, they can run in parallel without caring about the result of the other operation. In some cases, you will want to use the results of two operations that are *dependent* - while you still want to use the results of both at the same time, the second operation also needs the result of the first operation to work.

An example:

```
'use strict';

// ...

Promise.try(() => {
    return getDatabaseConnection();
}).then((databaseConnection) => {
    return databaseConnection.query("users", {id: req.body.id});
}).then((user) => {
    res.json(user);

    // This is not possible, because we don't have `databaseConnection` in scope anymore:
    databaseConnection.close();
});
```

In these cases, rather than using `Promise.all`, you'd *add a level of nesting* to keep something in scope:

```
'use strict';

// ...

Promise.try(() => {
    return getDatabaseConnection();
}).then((databaseConnection) => {
    // We nest here, so that `databaseConnection` remains in scope.

    return Promise.try(() => {
        return databaseConnection.query("users", {id: req.body.id});
    }).then((user) => {
        res.json(user);

        databaseConnection.close(); // Now it works!
    });
});
```

Of course, as with any kind of nesting, you should do it sparingly - and only when necessary for a situation like this. Splitting up your code into small functions, with each of them having a *single* responsibility, will prevent trouble with this.

Error handling (with Promises)

> This article was originally published at https://gist.github.com/joepie91/c8d8cc4e6c2b57889446. It only applies when using Promise chaining syntax; when you use `async`/`await`, you are instead expected to use `try`/`catch`, which unfortunately does not support error filtering.

There's roughly three types of errors:

1. **Expected errors** - eg. "URL is unreachable" for a link validity checker. You should handle these in your code at the top-most level where it is practical to do so.
2. **Unexpected errors** - eg. a bug in your code. These should crash your process (yes, really), they should be logged and ideally e-mailed to you, and you should fix them right away. You should never catch them for any purpose other than to log the error, and even then you should make the process crash.
3. **User-facing errors** - not really in the same category as the above two. While you can represent them with error objects (and it's often practical to do so), they're not really errors in the programming sense - rather, they're user feedback. When represented as error objects, these should only ever be handled at the top-most point of a request - in the case of Express, that would be the error-handling middleware that sends a HTTP status code and a response.

Would I still need to use try/catch if I use promises?

Sort of. Not the usual `try`/`catch`, but eg. Bluebird has a `.try` and `.catch` equivalent. It works like synchronous `try`/`catch`, though - errors are propagated upwards automatically so that you can handle them where appropriate.

Bluebird's `try` isn't identical to a standard JS `try` - it's more a 'start using Promises' thing, so that you can also wrap synchronous errors. That's the magic of Promises, really - they let you handle synchronous and asynchronous errors/values like they're one and the same thing.

Below is a relatively complex example, that uses a custom 'error filter' (predicate) function, because filesystem errors have a name but not a special error type. The error filtering is only available in Bluebird, by the way - 'native' Promises don't have the filtering.

```
/* UPDATED: This example has been changed to use the new object predicates, that were
 * introduced in Bluebird 3.0. If you are using Bluebird 2.x, you will need to use the
```

```
 * older example below, with the predicate function. */

var Promise = require("bluebird");
var fs = Promise.promisifyAll(require("fs"));

Promise.try(function(){
    return fs.readFileAsync("./config.json").then(JSON.parse);
}).catch({code: "ENOENT"}, function(err){
    /* Return an empty object. */
    return {};
}).then(function(config){
    /* `config` now either contains the JSON-parsed configuration file, or an empty object if no
configuration file existed. */
});
```

If you are still using Bluebird 2.x, you should use predicate functions instead:

```
/* This example is ONLY for Bluebird 2.x. When using Bluebird 3.0 or newer, you should
 * use the updated example above instead. */

var Promise = require("bluebird");
var fs = Promise.promisifyAll(require("fs"));

var NonExistentFilePredicate = function(err) {
    return (err.code === "ENOENT");
};

Promise.try(function(){
    return fs.readFileAsync("./config.json").then(JSON.parse);
}).catch(NonExistentFilePredicate, function(err){
    /* Return an empty object. */
    return {};
}).then(function(config){
    /* `config` now either contains the JSON-parsed configuration file, or an empty object if no
configuration file existed. */
});
```

Bluebird Promise.try using ES6 Promises

> This article was originally published at https://gist.github.com/joepie91/255250eeea8b94572a03.

Note that this will only be equivalent to `Promise.try` if your runtime or ES6 Promise shim correctly catches synchronous errors in Promise constructors.

If you are using the latest version of Node, this should be fine.

```
var Promise = require("es6-promise").Promise;

module.exports = function promiseTry(func) {
    return new Promise(function(resolve, reject) {
        resolve(func());
    })
}
```

Please don't include minified builds in your npm packages!

> This article was originally published at https://gist.github.com/joepie91/04cc8329df231ea3e262dffe3d41f848.

There's quite a few libraries on npm that not only include the regular build in their package, but also a minified build. While this may seem like a helpful addition to make the package more complete, it actually poses a real problem: it becomes very difficult to audit these libraries.

The problem

You've probably seen incidents like the `event-stream` incident, where a library was compromised in some way by an attacker. This sort of thing, also known as a "supply-chain attack", is starting to become more and more common - and it's something that developers need to protect themselves against.

One effective way to do so, is by auditing dependencies. Having at least a cursory look through every dependency in your dependency tree, to ensure that there's nothing sketchy in there. While it isn't going to be 100% perfect, it will detect most of these attacks - and not only is briefly reviewing dependencies *still* faster than reinventing your own wheels, it'll also give you more insight into how your application actually works under the hood.

But, there's a problem: a lot of packages include almost-duplicate builds, sometimes even minified ones. It's becoming increasingly common to see a separate CommonJS and ESM build, but in many cases there's a *minified* build included too. And those are basically impossible to audit! Even with a code beautifier, it's very difficult to understand what's really going on. But you can't ignore them either, because if they are a part of the package, then other code can require them. So you *have* to audit them.

There's a workaround for this, in the form of "reproducing" the build; taking the original (Git) repository for the package which only contains the original code and not the minified code, checking out the intended version, and then just running a build that *creates* the minified version, which you can then compare to the one on npm. If they match, then you can assume that you only need to audit the original source in the Git repo.

Or well, that *would* be the case, if it weren't possible for the *build tools* to introduce malicious code as well. Argh! Now you need to audit *all of the build tools being used* as well, at the specific versions that are being used by each dependency. Basically, you're now auditing hundreds of build

stacks. This is a massive waste of time for every developer who wants to make sure there's nothing sketchy in their dependencies!

All the while these minified builds don't really solve a problem. Which brings me to...

Why it's unnecessary to include minified builds

As a library author, you are going to be dealing with roughly two developer demographics:

1. Those who just want a file they can include as a `<script>` tag, so that they can use your library in their (often legacy) module-less code.
2. Those with a more modern development stack, including a package manager (npm) and often also build tooling.

For the first demographic, it makes a lot of sense to provide a pre-minified build, as they are going to directly include it in their site, and it should ideally be small. But, here's the rub: those are *also* the developers who probably aren't using (or don't want to use) a package manager like npm! There's not really a reason why their minified pre-build should exist *on npm*, specifically - you might just as well offer it as a separate download.

For the second demographic, a pre-minified build isn't really useful at all. They probably already have their own development stack that does minification (of their own code *and* dependencies), and so they simply won't be using your minified build.

In short: there's not really a point to having a minified build *in your npm package*.

The solution

Simply put: don't include minified files in your npm package - distribute them separately, instead. In most cases, you can just put it on your project's website, or even in the (Git) repository.

If you really do have some specific reason to need to distribute them through npm, at least put them in a *separate package* (eg. `yourpackage-minified`), so that only those who actually *use* the minified version need to add it to their dependency folder.

Ideally, try to only have a single copy of your code in your package at all - so also no separate CommonJS and ESM builds, for example. CommonJS works basically everywhere, and there's basically no reason to use ESM anyway, so this should be fine for most projects.

If you really *must* include an ESM version of your code, you should at least use a wrapping approach instead of duplicating the code (note that this can be a breaking change!). But if you can, please leave it out to make it easier for developers to understand what they are installing into their project!

Anyone should be able to audit and review their dependencies, not just large companies with deep pockets; and not including unnecessarily duplicated or obfuscated code into your packages will help a long way towards that. Thanks!

How to get the actual width of an element in jQuery, even with border-box: box-sizing

> This article was originally published at https://gist.github.com/joepie91/5ffffefbf24dcfdb4477.

This is ridiculous, but per the jQuery documentation:

> Note that `.width()` will always return the content width, regardless of the value of the CSS `box-sizing` property. As of jQuery 1.8, this may require retrieving the CSS width plus `box-sizing` property and then subtracting any potential border and padding on each element when the element has `box-sizing: border-box`. To avoid this penalty, use `.css( "width" )` rather than `.width()`.

```
function parsePx(input) {
	let match;

	if (match = /^([0-9+])px$/.exec(input)) {
		return parseFloat(match[1]);
	} else {
		throw new Error("Value is not in pixels!");
	}
}

$.prototype.actualWidth = function() {
	/* WTF, jQuery? */
	let isBorderBox = (this.css("box-sizing") === "border-box");
	let width = this.width();

	if (isBorderBox) {
		width = width
			+ parsePx(this.css("padding-left"))
```

```
    + parsePx(this.css("padding-right"))
    + parsePx(this.css("border-left-width"))
    + parsePx(this.css("border-right-width"));
}

return width;
}
```

A survey of unhandledRejection and rejectionHandled handlers

> This article was originally published at https://gist.github.com/joepie91/06cca7058a34398f168b08223b642162.

Bluebird (http://bluebirdjs.com/docs/api/error-management-configuration.html#global-rejection-events)

- `process.on//unhandledRejection`: **(Node.js)** Potentially unhandled rejection.
- `process.on//rejectionHandled`: **(Node.js)** Cancel unhandled rejection, it was handled anyway.
- `self.addEventListener//unhandledrejection`: **(WebWorkers)** Potentially unhandled rejection.
- `self.addEventListener//rejectionhandled`: **(WebWorkers)** Cancel unhandled rejection, it was handled anyway.
- `window.addEventListener//unhandledrejection`: **(Modern browsers, IE >= 9)** Potentially unhandled rejection.
- `window.addEventListener//rejectionhandled`: **(Modern browsers, IE >= 9)** Cancel unhandled rejection, it was handled anyway.
- `window.onunhandledrejection`: **(IE >= 6)** Potentially unhandled rejection.
- `window.onrejectionhandled`: **(IE >= 6)** Cancel unhandled rejection, it was handled anyway.

WhenJS (https://github.com/cujojs/when/blob/3.7.0/docs/debug-api.md)

- `process.on//unhandledRejection`: **(Node.js)** Potentially unhandled rejection.
- `process.on//rejectionHandled`: **(Node.js)** Cancel unhandled rejection, it was handled anyway.
- `window.addEventListener//unhandledRejection`: **(Modern browsers, IE >= 9)** Potentially unhandled rejection.
- `window.addEventListener//rejectionHandled`: **(Modern browsers, IE >= 9)** Cancel unhandled rejection, it was handled anyway.

Spec (https://gist.github.com/benjamingr/0237932cee84712951a2)

- `process.on//unhandledRejection`: **(Node.js)** Potentially unhandled rejection.
- `process.on//rejectionHandled`: **(Node.js)** Cancel unhandled rejection, it was handled anyway.

Spec (WHATWG: https://html.spec.whatwg.org/multipage/webappapis.html#unhandled-promise-rejections)

- `window.addEventListener//unhandledrejection`: **(Browsers)** Potentially unhandled rejection.
- `window.addEventListener//rejectionhandled`: **(Browsers)** Cancel unhandled rejection, it was handled anyway.
- `window.onunhandledrejection`: **(Browsers)** Potentially unhandled rejection.
- `window.onrejectionhandled`: **(Browsers)** Cancel unhandled rejection, it was handled anyway.

ES6 Promises in Node.js (https://nodejs.org/api/process.html#process_event_rejectionhandled onwards)

- `process.on//unhandledRejection`: Potentially unhandled rejection.
- `process.on//rejectionHandled`: Cancel unhandled rejection, it was handled anyway.

Yaku (https://github.com/ysmood/yaku#unhandled-rejection)

- `process.on//unhandledRejection`: **(Node.js)** Potentially unhandled rejection.
- `process.on//rejectionHandled`: **(Node.js)** Cancel unhandled rejection, it was handled anyway.
- `window.onunhandledrejection`: **(Browsers)** Potentially unhandled rejection.
- `window.onrejectionhandled`: **(Browsers)** Cancel unhandled rejection, it was handled anyway.

Quill.js glossary

> This article was originally published at https://gist.github.com/joepie91/46241ef1ce89c74958da0fdd7d04eb55.

Since Quill.js doesn't seem to document its strange jargon-y terms anywhere, here's a glossary that I've put together for it. No guarantees that it's correct! But I've done my best.

Quill - The WYSIWYG editor library

Parchment - The internal model used in Quill to implement the document tree

Scroll - A document, expressed as a tree, technically also a Blot (node) itself, specifically the root node

Blot - A node in the document tree

Block (Blot) - A block-level node

Inline (Blot) - An inline (formatting) node

Text (Blot) - A node that contains only(!) raw text contents

Break (Blot) - A node that contains nothing, used as a placeholder where there is no actual content

"a format" - A specific formatting attribute (width, height, is bold, ...)

`.format(...)` - The API method that is used to set a formatting attribute on some selection

Riot.js cheatsheet

> This article was originally published at https://gist.github.com/joepie91/ed3a267de70210b46fb06dd57077827a.

Component styling

> **This section only applies to Riot.js 2.x.** Since 3.x, all styles are scoped *by default* and you can simply add a `style` tag to your component.

1. You can use a `<style>` tag within your tag. This style tag is **applied globally** by default.
2. You can **scope your style tag** to limit its effect to the component that you've defined it in. Note that scoping is **based on the tag name**. There are two options:
3. Use the `scoped` attribute, eg. `<style scoped> ... </style>`
4. Use the `:scope` pseudo-selector, eg. `<style> :scope { ... } </style>`
5. You can change where global styles are 'injected' by having `<style type="riot"></style>` somewhere in your `<head>`. This is useful for eg. controlling what styles are overridden.

Mounting

"Mounting" is the act of attaching a custom tag's template and behaviour to a specific element in the DOM. The most common case is to mount all instances of a specific top-level tag, but there are more options:

1. Mount all custom tags on the page: `riot.mount("*")`
2. Mount all instances of a specific tag name: `riot.mount("app")`
3. Mount a tag with a specific ID: `riot.mount("#specific_element")`
4. Mount using a more complex selector: `riot.mount("foo, bar")`

Note that "child tags" (that is, custom tags that are specified within other custom tags) are automatically mounted as-needed. You do not need to `riot.mount` them separately.

The simplest example:

```
<script>
// Load the `app` tag's definition here somehow...

document.addEventListener("DOMContentLoaded", (event) => {
```

```
    riot.mount("app");
});
</script>


<app></app>
```

Tag logic

- **Conditionally add to DOM:** `<your-tag if="{ something === true }"> ... </your-tag>`
- **Conditionally display:** `<your-tag show="{ something === true }"> ... </your-tag>` (but the tag always *exists* in the DOM)
- **Conditionally hide:** `<your-tag hide="{ something === true }"> ... </your-tag>` (but the tag always *exists* in the DOM)
- **For-each loop:** `<your-tag for="{ item in items }"> ... (you can access 'item' from within the tag) ... </your-tag>` (one instance of `your-tag` for each `item` in `items`)
- **For-each loop of an object:** `<your-tag for="{ key, value in someObject }"> ... (you can access 'key' and 'value' from within the tag) ... </your-tag>` (this is *slow!*)

All of the above also work on *regular* (ie. non-Riot) HTML tags.

If you need to add/hide/display/loop a *group* of tags, rather than a single one, you can wrap them in a `<virtual>` pseudo-tag. This works with all of the above constructs. For example:

```
<virtual for="{item in items}">
    <label>{item.label}</label>
    <textarea>{item.defaultValue}</textarea>
</virtual>
```

Quick reference for `checkit` validators

> This article was originally published at https://gist.github.com/joepie91/cd107b3a566264b28a3494689d73e589.

Presence

- **exists -** The field must exist, and not be `undefined`.
- **required -** The field must exist, and not be `undefined`, `null` or an empty string.
- **empty -** The field must be some kind of "empty". Things that are considered "empty" are as follows:
  - `""` (empty string)
  - `[]` (empty array)
  - `{}` (empty object)
  - Other falsey values

Character set

- **alpha -** `a-z`, `A-Z`
- **alphaNumeric -** `a-z`, `A-Z`, `0-9`
- **alphaUnderscore -** `a-z`, `A-Z`, `0-9`, `_`
- **alphaDash -** `a-z`, `A-Z`, `0-9`, `_`, `-`

Value

Length-related validators may apply to both strings and arrays.

- **exactLength:** `length` **-** The value must have a length of exactly `length`.
- **minLength:** `length` **-** The value must have a length of at least `length`.
- **maxLength:** `length` **-** The value must have a length of at most `length`.
- **contains:** `needle` **-** The value must contain the specified `needle` (applies to both strings and arrays).
- **accepted -** Must be a value that indicates agreement - varies by language (defaulting to `en`):
  - **en, fr, nl -** `"yes"`, `"on"`, `"1"`, `1`, `"true"`, `true`
  - **es -** `"yes"`, `"on"`, `"1"`, `1`, `"true"`, `true`, `"si"`

    - **ru -** `"yes"`, `"on"`, `"1"`, `1`, `"true"`, `true`, `"да"`

Value (numbers)

Note that "numbers" refers to both Number-type values, and strings containing numeric values!

- **numeric -** Must be a finite numeric value of some sort.
- **integer -** Must be an integer value (either positive or negative).
- **natural -** Must be a natural number (ie. an integer value of 0 or higher).
- **naturalNonZero -** Must be a natural number, but *higher* than 0 (ie. an integer value of 1 or higher).
- **between:**`min`**:**`max` **-** The value must numerically be between the `min` and `max` values (exclusive).
- **range:**`min`**:**`max` **-** The value must numerically be *within* the `min` and `max` values (inclusive).
- **lessThan:**`maxValue` **-** The value must numerically be less than the specified `maxValue` (exclusive).
- **lessThanEqualTo:**`maxValue` **-** The value must numerically be less than *or equal to* the specified `maxValue` (inclusive).
- **greaterThan:**`minValue` **-** The value must numerically be greater than the specified `minValue` (exclusive).
- **greaterThanEqualTo:**`minValue` **-** The value must numerically be greater than *or equal to* the specified `minValue` (inclusive).

Relations to other fields

- **matchesField:**`field` **-** The value in this field must equal the value in the specified other `field`.
- **different:**`field` **-** The value in this field must *not* equal the value in the specified other `field`.

JavaScript types

- **NaN -** Must be `NaN`.
- **null -** Must be `null`
- **string -** Must be a `String`.
- **number -** Must be a `Number`.
- **array -** Must be an `Array`.
- **plainObject -** Must be a plain `object` (ie. object literal).
- **date -** Must be a `Date` object.
- **function -** Must be a `Function`.
- **regExp -** Must be a `RegExp` object.
- **arguments -** Must be an `arguments` object.

Format

- **email -** Must be a validly formatted e-mail address.
- **luhn -** Must be a validly formatted creditcard number (according to a Luhn regular expression).
- **url -** Must be a validly formatted URL.
- **ipv4 -** Must be a validly formatted IPv4 address.
- **ipv6 -** Must be a validly formatted IPv6 address.
- **uuid -** Must be a validly formatted UUID.
- **base64 -** Must be a validly formatted base64 string.

ES Modules are terrible, actually

> This post was originally published at https://gist.github.com/joepie91/bca2fda868c1e8b2c2caf76af7dfcad3, which was in turn adapted from an earlier Twitter thread.

It's incredible how many collective developer hours have been wasted on pushing through the turd that is ES Modules (often mistakenly called "ES6 Modules"). Causing a big ecosystem divide and massive tooling support issues, for... well, no reason, really. There are no actual advantages to it. At all.

It looks shiny and new and some libraries use it in their documentation without any explanation, so people assume that it's the new thing that must be used. And then I end up having to explain to them why, unlike CommonJS, it doesn't actually work everywhere yet, and may never do so. For example, you can't import ESM modules from a CommonJS file! (Update: I've released a module that works around this issue.)

And then there's Rollup, which apparently requires ESM to be used, at least to get things like treeshaking. Which then makes people believe that treeshaking is not possible with CommonJS modules. Well, it is - Rollup just chose not to support it.

And then there's Babel, which tried to transpile `import`/`export` to `require`/`module.exports`, sidestepping the ongoing effort of standardizing the module semantics for ESM, causing broken imports and `require("foo").default` nonsense and spec design issues all over the place.

And then people go "but you can use ESM in browsers without a build step!", apparently not realizing that that is an utterly useless feature because loading a full dependency tree over the network would be unreasonably and unavoidably slow - you'd need as many roundtrips as there are levels of depth in your dependency tree - and so you need some kind of build step anyway, eliminating this entire supposed benefit.

And then people go "well you can statically analyze it better!", apparently not realizing that ESM doesn't actually change any of the JS semantics other than the `import`/`export` syntax, and that the `import`/`export` statements are equally analyzable as top-level `require`/`module.exports`.

"*But in CommonJS you can use those elsewhere too, and that breaks static analyzers!*", I hear you say. Well, yes, absolutely. But that is inherent in dynamic imports, which by the way, ESM also supports with its dynamic `import()` syntax. So it doesn't solve that either! Any static analyzer still needs to deal with the case of dynamic imports *somehow* - it's just rearranging deck chairs on the Titanic.

And *then*, people go "but now we at least have a standard module system!", apparently not realizing that CommonJS was *literally that*, the result of an attempt to standardize the various competing module systems in JS. Which, against all odds, *actually succeeded*!

... and then promptly got destroyed by ESM, which reintroduced a split and all sorts of incompatibility in the ecosystem, rather than just importing some updated variant of CommonJS into the language specification, which would have sidestepped almost all of these issues.

And while the initial CommonJS standardization effort succeeded due to none of the competing module systems being in particularly widespread use yet, CommonJS is so ubiquitous in Javascript-land nowadays that it will never fully go away. Which means that runtimes will forever have to keep supporting two module systems, and **developers will forever be paying the cost of the interoperability issues between them**.

But it's the future!

Is it really? The vast majority of people who believe they're currently using ESM, aren't even actually doing so - they're feeding their entire codebase through Babel, which deftly converts all of those snazzy `import` and `export` statements back into CommonJS syntax. Which works. So what's the point of the new module system again, if it all works with CommonJS anyway?

And it gets worse; `import` and `export` are designed as special-cased statements. Aside from the obvious problem of needing to learn a special syntax (which doesn't *quite* work like object destructuring) instead of reusing core language concepts, this is also a downgrade from CommonJS' `require`, which is a *first-class expression* due to just being a function call.

That might sound irrelevant on the face of it, but it has very real consequences. For example, the following pattern is simply **not possible** with ESM:

```
const someInitializedModule = require("module-name")(someOptions);
```

Or how about this one? Also no longer possible:

```
const app = express();
// ...
app.use("/users", require("./routers/users"));
```

Having language features available as a first-class expression is one of the most desirable properties in language design; yet for some completely unclear reason, ESM proponents decided to *remove* that property. There's just no way anymore to directly combine an `import` statement with some other JS syntax, whether or not the module path is statically specified.

The only way around this is with `await import`, which would break the supposed static analyzer benefits, only work in async contexts, and even then require weird hacks with parentheses to make it work correctly.

It also means that you now need to make a choice: do you want to be able to use ESM-only dependencies, or do you want to have access to patterns like the above that help you keep your codebase maintainable? ESM or maintainability, your choice!

So, congratulations, ESM proponents. You've destroyed a successful userland specification, wasted many (hundreds of?) thousands of hours of collective developer time, many hours of my own personal unpaid time trying to support people with the fallout, and created ecosystem fragmentation that will never go away, in exchange for... fuck all.

This is a disaster, and the only remaining way I see to fix it is to stop trying to make ESM happen, and deprecate it in favour of some variant of CommonJS modules being absorbed into the spec. It's not too late *yet*; but at some point it will be.

A few notes on the "Gathering weak npm credentials" article

> This article was originally published in 2017 at https://gist.github.com/joepie91/828532657d23d512d76c1e68b101f436. Since then, npm has implemented 2FA support in the registry, and was acquired by Microsoft through Github.

Yesterday, an article was released that describes how one person could obtain access to enough packages on npm to affect 52% of the package installations in the Node.js ecosystem. Unfortunately, this has brought about some comments from readers that completely miss the mark, and that draw away attention from the real issue behind all this.

To be very clear: **This (security) issue was caused by 1) poor password management on the side of developers, 2) handing out unnecessary publish access to packages, and most of all 3) poor security on the side of the npm registry.**

With that being said, let's address some of the common claims. This is going to be slightly ranty, because to be honest I'm rather disappointed that otherwise competent infosec people distract from the underlying causes like this. All that's going to do is prevent this from getting fixed in *other* language package registries, which almost certainly suffer from the same issues.

"This is what you get when you use small dependencies, because there are such long dependency chains"

This is very unlikely to be a relevant factor here. Don't forget that a key part of the problem here is that publisher access is handed out unnecessarily; if the Node.js ecosystem were to consist of a few large dependencies (that everybody used) instead of many small ones (that are only used by those who actually need the entire dependency), you'd just end up with each large dependency being responsible for *a larger part of the 52%*.

There's a potential point of discussion in that a modular ecosystem means that more different groups of people are involved in the implementation of a given dependency, and that this could provide for a larger (human) attack surface; however, *this is a completely unexplored argument for which no data currently exists*, and this particular article does not provide sufficient evidence to show it to be true.

Perhaps not surprisingly, the "it's because of small dependencies" argument seems to come primarily from people who don't fully understand the Node.js dependency model and make a lot of (incorrect) assumptions about its consequences, and who appear to take every opportunity to blame things on "small dependencies" regardless of technical accuracy.

In short: No, this is not because of small dependencies. It would very likely happen with large dependencies as well.

"See, that's why you should always lock your dependency versions. This is why semantic versioning is bad."

Aside from semantic versioning being a practice that's separate from automatically updating based on a semver range, preventing automatic updates isn't going to prevent this issue either. The problem here is with *publish access to the modules*, which is a completely separate concern from "how the obtained access is misused".

In practice, most people who "lock dependency versions" seem to follow a practice of "automatically merge any update that doesn't break tests" - which really is no different from just letting semver ranges do their thing. Even if you *do* audit updates before you apply them (and let's be realistic, how many people *actually* do this for every update?), it would be trivial to subtly backdoor most of the affected packages due to their often aging and messy codebase, where one more bit of strange code doesn't really stand out.

The chances of locked dependencies preventing exploitation are close to zero. Even if you *do* audit your updates, it's relatively trivial for a competent developer to sneak by a backdoor. At the same time, "people not applying updates" is a far bigger security issue than audit-less dependency locking will solve.

All this applies to "vendoring in dependencies", too - vendoring in dependencies is no technically different from pinning a version/hash of a dependency.

In short: No, dependency locking will not prevent exploitation through this vector. Unless you have a strict auditing process (which you should, but many do not), you **should not** lock dependency versions.

"That's why you should be able to add a hash to your package.json, so that it verifies the integrity of the dependency.

This solves a completely different and almost unimportant problem. The only thing that a package hash will do, is assuring that everybody who installs the dependencies gets the exact same dependencies (for a locked set of versions). However, the npm registry *already does that* - it prevents republishing different code under an already-used version number, and even with publisher access you cannot bypass that.

Package hashes also give you absolutely zero assurances about future updates; *package hashes are not signatures*.

In short: This just doesn't even have anything to do with the credentials issue. It's totally unrelated.

"See? This is why Node.js is bad."

Unfortunately plenty of people are conveniently using this article as an excuse to complain about Node.js (because that's apparently the hip thing to do?), without bothering to understand what happened. Very simply put: **this issue is not in any way specific to Node.js.** The issue here is an issue of developers with poor password policies and poor registry access controls. It just so happens that the research was done on npm.

As far as I am aware, this kind of research has not been carried out for *any* other language package registries - but many other registries appear to be similarly poorly monitored and secured, and are very likely to be subject to the exact same attack.

If you're using this as an excuse to complain about Node.js, without bothering to understand the issue well enough to realize that it's a *language-independent issue*, then perhaps you should reconsider exactly how well-informed your point of view of Node.js (or other tools, for that matter) really is. Instead, you should take this as a lesson and *prevent this from happening in other language ecosystems*.

In short: This has absolutely nothing to do with Node.js specifically. That's just where the research happens to be done. Take the advice and start looking at other language package registries, to ensure they are not vulnerable to this either.

So then how should I fix this?

1. Demand from npm Inc. that they prioritize implementing 2FA immediately, actively monitor for incidents like this, and generally implement all the mitigations suggested in the article. It's really not reasonable how poorly monitored or secured the registry is, especially given that it's *operated by a commercial organization*, and it's been around for a *long* time.
2. If you have an npm account, follow the instructions here.
3. Carry out or encourage the same kind of research on the package registry for *your* favorite language. It's very likely that other package registries are similarly insecure and

poorly monitored.

Unfortunately, as a mere consumer of packages, there's nothing you can do about this other than demanding that npm Inc. gets their registry security in order. This is fundamentally an infrastructure problem.